Hadoop Independent Study

Elliott Fodi

Purpose:

The purpose of this independent study was to learn Hadoop and frameworks related to Hadoop. With the majority of large companies using Hadoop for large scale processing and data mining, knowing Hadoop is a quality employers are looking for. In all of my classes, I heard about Hadoop, and how it is extremely power and can be used to solve large problems but I never had the opportunity to work with Hadoop. It is one thing to know about the framework and another to work with it. Hadoop is a vast framework, that covers many different areas of computer science. Covering every aspect of Hadoop is unfeasible in a semester. I choose to learn only the important aspects of Hadoop like creating a personal cluster for testing code, writing MapReduce programs, and utilizing HDFS for other frameworks (HBase). While learning Hadoop it was my goal to create a programs and guides for students that also wish to learn Hadoop. My hopes are that Hadoop will become part of the RIT curriculum, since it currently is not.

Brief explanation of Hadoop:

Hadoop is a framework for large scale distributed computing. Based off (GFS), it is designed to run on commodity hardware while providing fault tolerance and scalability. Hadoop is capable of scaling from one node to thousands of nodes; it is Hadoop's large scalability that makes it extremely powerful, with the ability to process large amounts of data relatively quickly, turning useless data into useful information. Hadoop was originally designed to be run on a cluster and perform batch processing. This means processing a large amount of data that is already sitting in the file system. Due to Hadoop's success it gained popularity and people started using it in unintended ways, leading to the development of new frameworks and modifications to the original framework.

Installing Hadoop:

The first part of my independent study was spent on installing Hadoop. Hadoop runs on Mac and Linux distributions. I choose to install it on Ubuntu 14.04 LTS as there is a large support community for running Hadoop on Ubuntu. Before installing Hadoop, Java must be installed, preferably the latest version. Hadoop requires that the JAVA_HOME variable be set, which can be done in the .bashrc file. After setting JAVA_HOME, Hadoop is ready to be installed. It can be downloaded from apache Hadoop's website where there are several different versions to choose from. Hadoop 1.0.0 releases work on the old framework which paired MapReduce and Hadoop Distributed File System (HDFS). The two components were not separable, which meant the trackers used to keep track of jobs and the file system were combine with the MapReduce framework. With the release of Hadoop 2.0.0 these trackers are no longer attached to the MapReduce. Hadoop 2.0.0 uses Yet Another Resource Manager (YARN) to manage all of the job tracking. This means using frameworks like HBase and Hive now sit on top of YARN and utilize the same HDFS file system. YARN allows for multiple frameworks to run on top of HDFS at the same time, it can be thought of as a general platform that can be

used to host various different farm works. It is important to note these differences in the version because this can be a large source of confusion.

When learning Hadoop it is best to start with 2.0 since it is the latest most widely adopted version. Most of all the external frameworks that utilize HDFS have been converted to work with YARN. Although if working with a older framework or an older version of a framework, Hadoop 1.0.0 may be needed. Hadoop 1.0.0's configuration files are set up differently from 2.0.0's config files. Also 1.0.0 does not support all the MapReduce features that 2.0.0 does. So it is important to know which version you need or which version you will be working with. Also even if you choose the latest version of Hadoop it will be out of date within a month. Since Hadoop is so widely used new version are always being released. Through the course of my independent study I started with Hadoop 2.5.0 in September and Hadoop 2.6.0 just released on November 30th. When learning Hadoop these changes are usually minor and using the latest stable version will be adequate for about a year. After a year the instillation should be upgraded. Once a version has been selected, Hadoop's website states downloading the tar.gz option is the best for beginners. Although a src option is available as well, this is suggested for only users that need the Hadoop src files.

Now that Hadoop has been downloaded the installation can begin. Unzip the tar ball and move the files to a directory of your choosing. At this point there are several different configuration (config) files that will need to be altered based on the mode Hadoop will be running in. Hadoop has three different modes it can be run in : stand alone, pseudo distributed, and fully distributed mode.

Stand alone mode- This is a standalone installation of Hadoop, meaning it does not use HDFS, all files are stored on the local file system. All of the Daemons are not running to reduce overhead, and since HDFS is not running, all of the HDFS daemons are off as well. This mode is meant for learning purposes only, the installation is treated as a single node cluster, where the master and the slave work on the same machine. Basically this mode is meant to easily and quickly test MapReduce programs. It is said that some features may not work in this mode since HDFS is not utilized. Most people prefer not to work with this as it is a watered down instillation and programs that may run fine in this mode, may have unseen problems running in a distributed mode. This is the default mode that Hadoop is configured for when unpackaged.

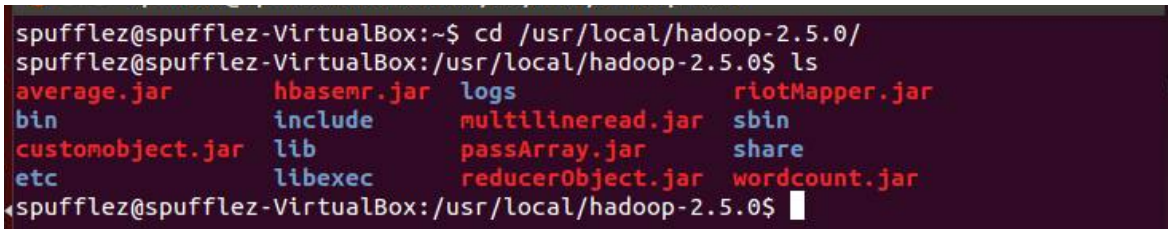
Pseudo distributed mode- This mode is designed to be run on one computer like standalone mode but it is a full running instillation of Hadoop. Here there is a master and one slave node both running on the same machine. All of the daemons are running between the master and the slave. HDFS is also working in pseudo distributed mode. There are a few more changes that need to be made to the config files to get pseudo distributed mode working but these are minor. This is in my opinion the best mode to learn Hadoop, as it gives the user all the expected features of a cluster and it runs on a single machine. Programs tested and designed

to run on pseudo distributed mode should have no problems running on fully distributed mode as they are the same. The only difference is there is one slave node versus multiple slaves.

Fully distributed mode- As described in its name this is the complete framework designed to be run with multiple slaves, each on their own machine, across a network. This is the installation that would be installed on a regular Hadoop cluster. This does how ever call for the most config files to be edited. Since there are multiple network connections that the master needs to deal with, most of the config files need network information added. It is possible to create pseudo distributed mode out of fully distributed mode but with all the additional config files that need to be edited, it is not worth the time. This is the mode that would be used to link slaves to master that are not not on the same computer.

Depending on the hardware available and the type of installation being performed, this will decide which mode to install. Each mode requires different changes be made to the config files but they do build on each other. So if standalone mode is installed it can easily be converted to pseudo distributed mode with a few additional changes. For learning Hadoop pseudo distributed mode is the best because it provides all the functionality of a regular cluster while still being able to run on a single machine.

As mentioned above there are several configuration files that need to be edited in order for Hadoop to run. Not being accustomed to doing system administration on linux, is where I had the most difficulty. To start, knowing the difference between the different versions of Hadoop is key. There is a lot of old and outdated documentation on the internet that will run with one version of Hadoop but not the other. Even some of the documentation on Hadoop's web site is outdated or not used in practice. When working with Hadoop 2.0.0 the config files will include configurations for YARN which if used in 1.0.0 files will not work. Also there is 1.0.0 configurations that will not work with 2.0.0 files. I found searching for your problem and adding the version of Hadoop you are working with leads to the best results. Just searching for the problem, leads to 1.0.0 solutions.



Caption: This shows the Hadoop directories. The blue files are the Hadoop directories and the red file are MapReduce programs I was working on.

The main config files that need to be edited are:

core-site.xml- This is where the file system is named. Since Hadoop is being run in pseudo distributed mode the local host is used as the master (namenode) for HDFS. By setting this,

Hadoop can now let all of its slave nodes know where the master node is for the file system is located. Further file system configurations will be made in the hdfs-site.xml file.

hadoop-env.sh- this contains core configuration options for Hadoop. This is where JAVA_HOME is set. Aside from setting the JAVA_HOME variable there is nothing else that needs to be edited for stand alone and pseudo distributed mode. Apache suggests to edit the JAVA_HOME variable in this file because then all other slave nodes will be using the same path. Other configuration options can be changed in here, like where logs are stored, heap size, run time options and more.

hdfs-site.xml- This is where changes to file system would be made. Here the replication factor is set and the paths on the local file system where Hadoop can store its HDFS files. For pseudo distributed mode the replication factor should be set to one because there is only one node to store files on. For a fully distributed mode leaving this the default three would be sufficient. It is important to set the paths where files on the HDFS will be saved. If these are not set the default saves these file to the temp directory. If the cluster is brought down for any reason, the file system is lost. These paths need to be configured for the master and the slave nodes.

mapred-site.xml- This is where all the MapReduce options are set. How ever if using Hadoop 2.0.0 the only changes that needs to be made here is to point MapReduce at yarn. If using an older version of MapReduce other options will need to be set.

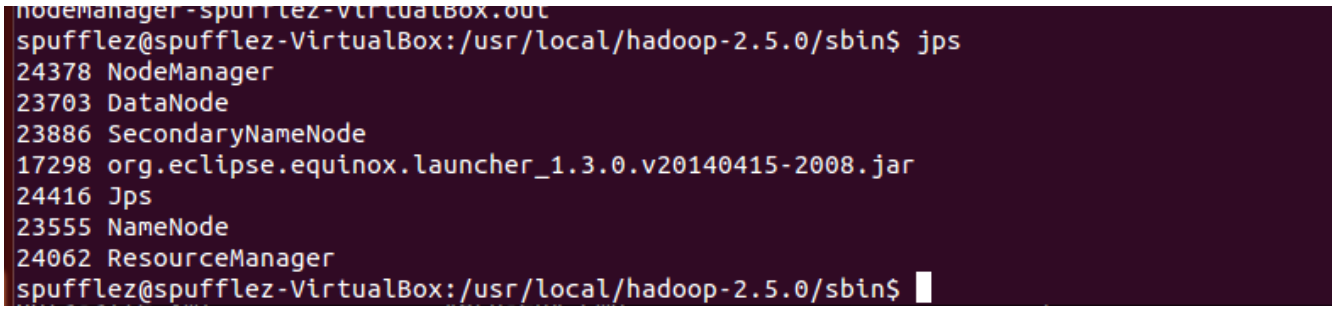
yarn-site.xml- This is where configurations to MapReduce sort options can be changed. These changes apply to the sorting that happens to all the key value pairs that mappers emit. Foar bsic installs setting these options to shuffle will work for most jobs.

It should be noted that each of the site files overrides the default values. So you only need to configure the options you want to change, everything else will stay as the default. These files can cause the most problems because if one option is mis configured or configured to the wrong value the entire framework will throw errors. I spent countless hours battling with these config files and editing different options before I could get Hadoop to run. For people who just want to learn MapReduce starting on a working cluster would be the best. Although it is good to know about these file incase some of the more complex features of Hadoop are needed.

Aside from editing the config files thats all the configuration needed for a basic Hadoop installation. What I learned over the course of my independent study was, the running version is very important when it comes to Hadoop. Since Hadoop is so popular things are always changing and new versions are popping up constantly, sometimes several times within a month. Keeping track of all changes is time consuming and requires a person dedicated to Hadoop administration.

Now that Hadoop is installed there are several ways to test the installation. First by trying to start the cluster. If there is misconfigured line in a config file, the cluster will attempt to start and throw an error. Whats even more frustrating is when it does not throw an error. There are cases when a node will fail to start but all the other nodes will start. This won't throw an error as Hadoop will attempt to start the node several times before it gives up. Using the jps command will reveal the java process running. The jps command is the quickest, most effective way to check what is running on the cluster. I used it countless times to debug issues with failed name nodes, slave nodes, and managers. When a node failed to start and I did not know why, I used the Hadoop log files to figure out the issue. Hadoop produces log files for every node and their managers. Hadoop logs everything so it is important that you look at the correct log file. The newest entry is append to the bottom of the log file. When errors occur this can cause other nodes to throw errors, so it takes time to resolve issues.

When starting a Hadoop cluster there are several approaches to this as well. Each daemon can be started individually or prepackaged scripts can be run to start the various daemons. The file system must be started first then the managers. When starting the file system the jps command will show namenode, datanode, and secondary namenode. If all of these are running then start the managers using YARN. The jps command will display resource manager and data manager. If all of these services are running then the cluster should be available for use. To further test the cluster, Hadoop comes with pre packaged test programs which can be run to test for errors. If the test program is successful, then no further changes need to be done.



Caption: JPS command showing the HDFS and YARN processes running

Brief explanation of HDFS:
Hadoop Distributed File System (HDFS) is a specialized file system designed to provide fault tolerance and access to data at any time. Since this file system is run on commodity hardware, failures will happen. When a file is added to HDFS, the file is broken into chunks and split across multiple nodes. Each chunk is then replicated three times by default to other nodes. This way, when a node fails the data is still accessible.

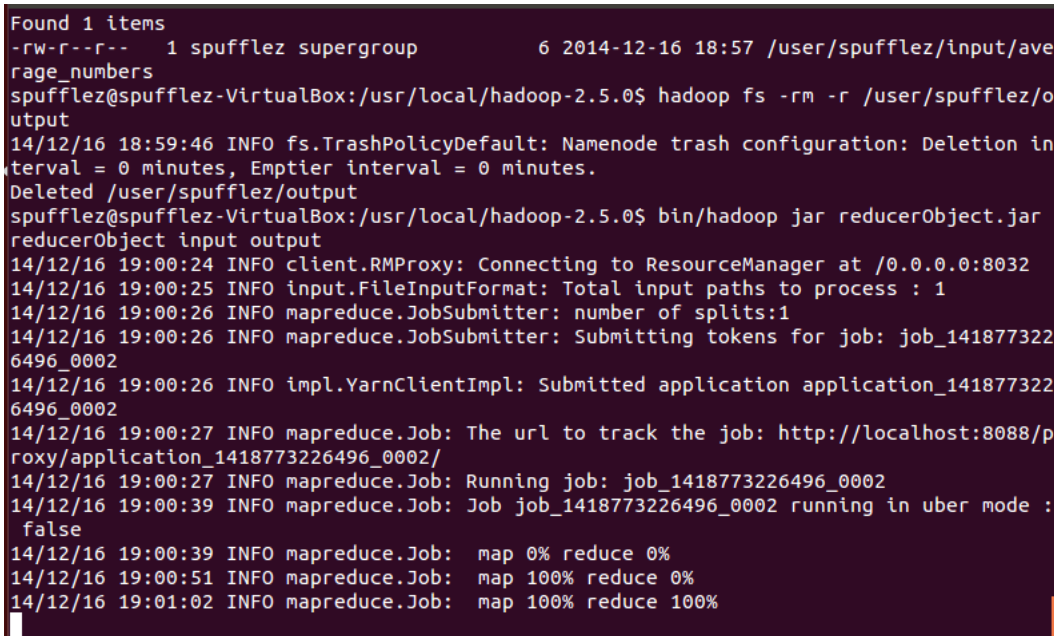
Using HDFS:
In Order to run a MapReduce program on Hadoop I first needed to learn how HDFS worked, which provided several challenges. I needed to figure out the commands to do things in the

file system. Hadoop uses commands similar to standard linux commands. The difference is the word “Hadoop fs” must be used before the command is issued and each command starts with a “-“. Other than that, for what I needed, all of the normal linux cli commands worked, like “ls”, “mkdir”, “cat” and more. When starting HDFS for the first time the file system needs to be formatted. After being formatted, the first thing is to create directories. Hadoop requires you to make each directory individually. Passing Hadoop a path like “/user/name/input” will fail if the directories user and name were not already created. Also when creating directories it is best practice to create a user directory first and then place all the users by name in that directory. When using the -ls command to list the contents of a directory the full path to the directory must be provided or -ls will return the contents of the root directory. Due to the file system being distributed the full paths to files and directories must be used each time.

Brief explanation of Map reduce:
MapReduce, is a method for solving large complex problems in a distributed manner. The data provided is split into records; each mapper receives a record and performs some operations on it. It then emits the result of its operation in the form of a key/value pair. The reducer then combines all these key/value pairs together and writes the final results to a file. This process works well for distributed computing because it allows for nodes to run mappers and reducers based on where the data is located or availability.

Creating a MapReduce program:
The Hello World of MapReduce is word count. This is where mappers are each assigned a different line of a file and they emit the word and a count of how many times it appeared. The reducer will then sum the different key/value pairs, print each word and its total count. There are several examples of this online and on Hadoop’s web site. While the concept is not hard to comprehend writing the program from scratch has its challenges if you haven’t seen Hadoop’s syntax before. Before even writing the program I faced an unusual challenge, locating the jar files needed to import the various Hadoop classes. On Hadoop’s site it gives detailed instructions on how to compile and run MapReduce programs from scratch. There are several other sites that give nice explanations on how to compile MapReduce programs. The problem is, none of these sites state where the jar files are, and searching for an individual jar file online will bring up the documentation, listing all the classes it contains. After searching through all the directories that comes with Hadoop I finally found them. Hadoop packages its jar files in a share folder under Hadoop. All of the jars that are compatible with the installed version of Hadoop are available here. Although if you want to know what jar files you need to include in a MapReduce program, Hadoop’s documentation must be consulted. I found that there are three main files that need to be included in the class path when writing a MapReduce program: Hadoop commons, common cli, and mapreduce core. These files will give access to all of the basic MapReduce features. For the word count program, on Hadoop’s site without any alterations. The program has a mapper, reducer, and a main which work as a great template for building future MapReduce programs. Later I will discuss the other MapReduce programs I wrote in detail. To run the program it must first be jared. In order to do this, compile the program using a standard java compiler, make sure to include the

Hadoop MapReduce jars in the class path. Hadoop will throw errors if something is wrong in the code which can later be debugged.



Caption: This shows the MapReduce program running, at this stage it is fully completed, as 100% appears for both the mapper and the reducer.

Main points about MapReduce programs:

Main

configuration- the base class that is configurable. Hadoop configuration is created using the values in the Hadoop xml files.
job- this is the job that will be run on the cluster which utilizes the Hadoop configuration.
set jar- tells Hadoop the name of the main
set mapper class- tells Hadoop the name of the mapper
set reducer class- tells Hadoop the name of the reducer
set output key class- is the type of key emitted by the mapper and the reducer
set output value class- is the type of value emitted by the mapper and the reducer

Mapper and reducer

The mapper will accept a record, until an emit is issued, the mapper is composed of plain java code.

context- is how the mapper and reducer emit key value pairs.

If the mapper or reducer's inputs or outputs do not match what is stated in the main, the program will throw an error. One of the problems I ran across was the main had different values set for the mapper to output and the reducer to accept as input. It is crucial that these match. It is also possible for the mapper to emit different values then the reducer. If this is to

be done there is an option available, to set what the mapper will emit, which will override the set output key/value for the mapper.

Hadoop uses its own special primitives to pass values from the mapper to the reducer. This is because the values need to be serialized when transversing the network and deserialized when received at the reducer. The keys need to be of the same format for Hadoop to sort them and send them to the correct reducer. These special primitives are of type Writable, meaning it can be written out of the mapper. Hadoop has Writable classes for all the basic java primitives. For example an int is now an intWritable and a long is now a longWritable. Strings are the only primitive that does not follow this naming convention, writable Strings are of type Text.

MapReduce Programs:

Passing a jar as an argument

For this map reduce program the goal was to pass an external jar file as an argument to Hadoop. This jar was to be used in the mapper or reducer for a calculation. In my example I used a jar that parsed JSON in the mapper. If using the jar multiple times it should be added to the list of jars that Hadoop loads by default. Although if the jar is only going to be used once it is not necessary to added to Hadoop's library. To pass a jar as an argument the MapReduce program needs to use tool runner. There are a few changes that need to be done for this to work. The body of the main is now implemented in the Tool Runners run method. The only code in the main, is to create and start a tool runner. Without this, when the option to pass a jar is given in the command line, Hadoop will not recognize it. As the name suggests this is used for more than just passing external jar files. Tool runner allows for an assortment of command line options to be given. This would be similar to using the -classpath command to add a class path when compiling a java program. This the same concept except the MapReduce main needs to parse and handle the arguments.

Passing an array to the reducer

In all of the basic examples online and in books, the mapper always outputs one value per key value pair, to the reducer. I wanted to have the mapper output several values to the reducer. This would reduce the amount of times the mapper must emit and reduce the sorting required for the reducer. To accomplish this I decided to pass an array from the mapper to the reducer. To pass an array from the mapper to the reducer, the ArrayWritable class is used. The class must be extended and the type of value being stored must be specified using the super command. Once done the new class can be used to pass arrays from the mapper to the reducer. Multiple values can be sent to the reducer via Maps as well. Hadoop has its own MapWritable class which is the equivalent of a Hash Map. Depending not the use there are multiple options available including custom objects covered in a later program.

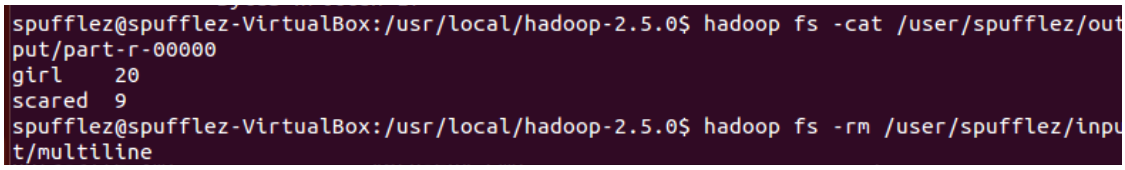
Taking an average

The reason for doing an average was to demonstrate how to store values and use them later in the reducer after all the key value pairs have been processed. This would be like using a

global variable, accessible anywhere in the reducer. These variables are declared outside the reduce method but can be accessed anywhere in the reduce method. Also other functions could be declared in the reducer which could be used in the reduce method as well. The purpose of this program was to do a simple demonstration of this. Note the reduce method is inside the reducer class.

Reading multiple lines as a record

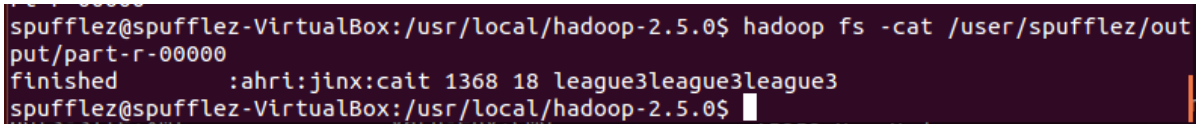
Hadoop by default will send one line of a file to each mapper. It sends these lines in the form of a key value pair where they key, which is most often discarded and not used is the line offset and the value is the text of the line. The goal of this program was to change the number of lines sent to each mapper. If processing files where each record is composed of two lines, the mapper needs to receive both of those lines. To do this several classes need to be changes. First, is how the lines are read, this is done by extending Hadoop's Text Input Format class and calling a custom record reader. The second part is to create this custom record reader. This class deals with the parsing and splitting up of the file to the mapper. The custom record reader extends record reader and is required to implement seven methods. The initialize and next Key Value are where all the work happens. Initialize sets up the reader and tells it where to start reading. If every file needs the first 3 lines skipped, this is where that would set. For reading multiple lines, this method does not need to be edited much from the default Hadoop initializer. The next key value method does need to be edited. This is where the reader is configured to read multiple lines at a time. It should be noted that this is also where the functionality to deal with the end of a file is taken care of as well. Finally the main class needs to be edited to accept this new reader. Using the set input format job option allows Hadoop to use the new input format created. As mentioned above this is very useful when dealing with files that need multiple lines but it can also be used on entire files as well. The same record reader can be modified to read an entire file instead of a few lines. This way the files are not split by lines, the mapper receives the whole file to process. Depending on the data this can be a useful feature.



Caption: This shows the results of running the multiline read program. The lines passed were;
wow are those on fire?
yes, are you scared
no I dont want the flames to catch my hair on fire
and there you have it, go ahead girl
The program will output the last word of the record received and the amount of words in the record. As shown the answer is correct.

Passing an object from mapper to the reducer

What happens when the mapper needs to output several different values, like names, ages, and addresses? One option is to create a multi purpose reducer that can handle different values emitted from the mapper. Another option is to have the mapper output an object and let every reducer be the same, sticking to the standard MapReduce model. When emitting a custom object, the object needs to implement the writable class. This is the class used for serialization. The class requires two constructors and read/write methods. If the primitives that compose the class are Hadoop writables the read and write methods are easy to write. Each primitive calls their respective readFields or write methods. If the objects are not Hadoop primitives then, convert it to a Hadoop primitive or create a writable version of the object. Once the object has been passed from the mapper to the reducer I found it easier to further convert the writable object back into a java object and use the java object in the reducer. Although it is not necessary, the toString method can be overwritten as well to provide for correct printing if the object is not being passed to a reducer. If this is not implemented the mapper will write the memory location.



Caption: This shows the output of the “pass object to reducer program”. The contents of the object passed from the mapper to the reducer were 3 names, 3 numbers, numbers filling an array, and the word league with the key appended to it. The output is all of the values from each object passed added or concatenated together, which is seen above.

Reducer outputting a non writable object

When passing an object from a mapper to a reducer the object needs to implement the writable class. Although when emitting an object from the reducer to HDFS it does not need to implement the writable class. Since the reducers write directly to HDFS and HDFS does not need data to be serialized, a java object can be emitted from the reducer. Again the only catch with this is that a toString method must be provided.

Lessons Learned:

Upon learning how to use Hadoop and write MapReduce programs I was faced with a project where I could apply what I had learned. While the results of this project were not part of my independent study, the reason I was able to use Hadoop for this project was because of my independent study. As a part of my independent study I am analyzing this program. A known fact is that Hadoop is horrible with small files. In this project I parsed thousands of small files using a Hadoop cluster. As reported in several papers the overhead involved in parsing these files, significantly impacted Hadoop’s performance. When parsing a single file using a java program the entire program ran in less then a second but when running the same parser on a 15 node Hadoop cluster the program took the same amount of time as if a single computer

were parsing 5000 files. There was no speedup due to the amount of overhead Hadoop generates. As a result I learned using Hadoop for any small jobs is a waste of money. Each node needs to be processing for several minutes to justify using Hadoop.

Since Hadoop programs can be painful to debug one way to make this process easier, is to write everything in java first. Since the mapper and the reducer run java code, writing the mapper and reducer before adding any Hadoop syntax saves an enormous amount of time debugging. With properly working java code it is much easier to integrate a java program into Hadoop versus starting from scratch. This also makes it easier to debug the Hadoop program because you can test the Hadoop components with bogus values which eliminates more errors. Overall the lack of having printout statements makes debugging Hadoop programs challenging in general.

Another tip to help debug Hadoop programs which I overlooked until late in the semester, is to run the mapper without a reducer. This may require a custom toString method depending on what the mapper is outputting. By doing this you can see exactly what the mapper is out putting. The program should be run with a small input because the mapper can generate a large output. This also helps with situations where the mapper completes and the reducer fails. Knowing what the reducer is accepting from the mapper will help resolve issues in the reducer.

While it is generally assumed, if the program runs correctly with a small amount of records it will do ok with a large amount of records, generally this is true, but it is not always the case. I ran into an instance where the reducer was at risk of running out of memory. The machines running the reduce tasks needed to have their minimum amount of memory increased in order to store all the values. This is not be apparent when using a small amount of records. Also dealing with corrupted records can cause a slew of issues. Most of the time Hadoop is used to analyze millions of records which means, there will be some corrupt records. If the mapper does not know how to deal with these corrupt records it can kill the entire MapReduce program. This becomes an issue if the program has been running for several hours and is just about to complete, when it encounters a corrupt or bad record, causing the MapReduce program terminate. There are a few ways to deal with this problem. One is to set Hadoop to skip records after a specified amount of attempts. This works great for corrupted files but what if the file is 99% intact and the data is still usable. The only way to deal with this is to write a mapper that can deal with all situations. This can be very tedious to write since it is difficult to think of all the different scenarios where the program might fail or where the file might be corrupt. When running MapReduce programs on large data sets it is extremely important to handle these cases or money will be wasted on failed attempts.

This is by far the most challenging part of writing a MapReduce program. Simply because the reducer needs to aggregate all of the data emitted from the mappers. If any sort of calculations need to be performed on this data, the reducer needs to store all the data. This can cause memory and storage issues.

Hadoop is not the answer to all problems, it is important to know when to use Hadoop and when to use a single node java program. The overheads involved with Hadoop make it an unattractive solution for smaller data sets or problems. Also the cost of maintaining and using a Hadoop cluster is not cheap. Using a cluster hosted on Amazon can be quite costly after several hours of use. In my opinion people are told about Hadoop and how useful it is, which paints a false image of the frame work. People gain the impression that Hadoop can solve all their problems. This is not the case, after using it for a semester it is designed for a specific type of problems, but most people to not realize this because they have not used it. The phrase just use Hadoop is thrown around too lightly. Once people use Hadoop and see what it is really capable of I feel they will have a better understanding of where to use it.

HDFS is the most valuable component of Hadoop. While MapReduce is a power feature, almost every other data processing framework has a version of MapReduce but they don't have a file system. There are multiple frameworks that run on Hadoop solely because of HDFS. The file system provides excellent fault tolerance and data availability which is what most systems require. Moving MapReduce to run on YARN was definitely a step in the right direction, with out it I do not think Hadoop would be as versatile as it is today. The more that can be done to make HDFS easier to use the better.

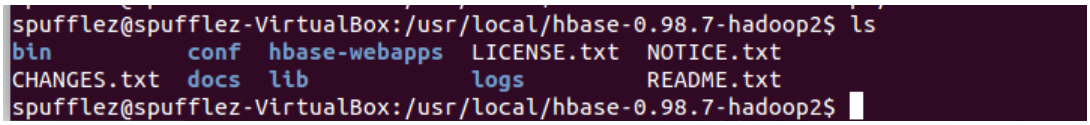
HBase

HBase is a NoSQL database that sits on top of HDFS. It provides indexing for large amounts of data and allows for MapReduce programs to be run on its tables. HBase is an open source Apache project, meaning anyone can freely use it and anyone can contribute to it. The system is written in Java, but this is not the language that needs to be used when interfacing with it. There are api's for thrift and REST that can be used instead of Java. Just like Hadoop has a name node and data nodes, HBase is structured in a similar way. Name nodes are Hmaster nodes and data nodes are region nodes which means if the master fails the system will not be usable until the master is replaced. [3]

Knowing how HBase stores its data will allow for better table creation because it sorts each record based on the row. This can be best seen with an example. For the following files file1.txt, test.txt, file3.pdf, depending on the use of the database, indexing these files like txt.file1, txt.test, pdf.file3, would provide for a fast look up for .txt files. If the filename is more important, than indexing by the name would place file1.txt, file3.pdf, before test.txt. Since HBase is sorted by the row indexing is very important and can mean the difference between a slow look up and a fast look up. The best way to visualize how HBase stores its data is in JSON format because some rows may or may not contain values for a column. If a row does not contain a value for a column, then nothing is stored. This approach allows for a better use of disk space and allows the database to hold more information.

Installing HBase:

Installing HBase is actually very easy, as the documentation on the HBase website is extremely accurate. That being said it is easy if the Hadoop cluster it is being run on is configured correctly. Since the documentation was very useful, I did not make my own and I will not cover it in detail because it is all freely available on there web site. In general the installation is similar to Hadoop in that after downloading the HBase files the JAVA_HOME variable needs to be set. Also HBase needs to be directed towards the Hadoop installation. One of the problems I did come across was, if the default port number for the Hadoop master has changed, use the changed port number. If the default Hadoop port is not in use, HBase will fail to connect and issue an error, master not found. That aside the installation is very simple.



Caption: This shows the HBase directories.

Caption: This shows the output of the jps command, showing the HMaster, HRegionServer, and HQuorumPeer running.

HBase cli:
While most people will use a program to manipulate data in HBase, there is a command line option available. The command line has all the available features that a program would have. The cli uses specific HBase commands to create, edit, and view tables. It is a great learning tool because it quickly allows the user to visually check what is in HBase. The usefulness of this may dwindle when working with larger data sets, as displaying millions of records would not be helpful.

Describing the data model:
HBase uses a different method of storing data from traditional databases. Traditional databases are row oriented, HBase is column oriented. This means accessing and inserting data is done differently. Tables are composed of rows and column families. Each row has a row key which is a unique value used to identify corresponding data. Column families are like groupings of columns, the purpose of a column family is to store similar data, physically close to each other. Inside a column family there are column qualifiers which are the actual columns

that store data. Each column qualifier starts with the name of the column family it belongs to, a colon, and then the column qualifier name. An example may make things clearer. Each row is assigned a timestamp, when data is written a new timestamp is appended to the data. Think of a time stamp as sub rows with in the row key. When a request for data is made the data contained in the most recent timestamp is returned. If a timestamp is provided when requesting data (the user wants to retrieve data at a specific time) then data corresponding to the time stamp provided is returned. Since the data is column oriented the data can be stored differently from a traditional database. A traditional database will store data based on rows and if a row does not contain a value for a column it is left blank, this blank space is wasted space. In a column oriented database the data is stored in a JSON like format. Meaning if a value for a column is null, there is no blank space. JSON is the easiest way to visualize this, as the column qualifier will exist but there is no blank cell, wasting space. This allows HBase to store more data, in an easy to retrieve way.

This example uses cars.
The car name is the row key, since car names are unique.
The column family is called car stats. In this case there is only one column family but there can be more if needed.
The column qualifiers are: make, year and Brake Horse Power (BHP)

Row Key	Timestamp	Column Family 1		Column Family 2	Column Family 3
		Column Qualifier 1-1	Column Qualifier 1-2	Column Qualifier 2	Column Qualifier 3
rk 1	t1	v1			
rk 2	t2		v2		
rk 3	t3			v3	
rk 4	t4				v4
	t5				v5

Table: Visual representation of how HBase stores its data

JSON view
Table Cars
row1 one:1
column family carStats
column qualifier carStats:make
Koenigsegg
column qualifier carStats:year
2014
column qualifier carStats:BHP
1341

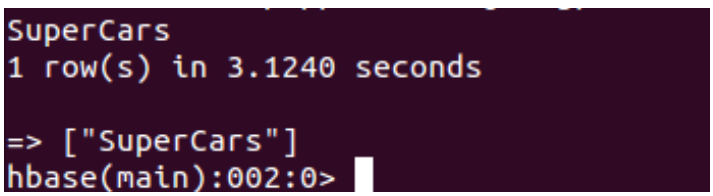
row2 one77
column family carStats

column qualifier carStats:make
Aston Martin
column qualifier carStats:year
2009
column qualifier carStats:BHP
750

row3 GTR
column family carStats
column qualifier carStats:make
Nissan
column qualifier carStats:year
2013
column qualifier carStats:BHP
600

Adding tables:

Tables can be added in two ways, by the cli or by a program. Hbase allows for programs to be written in a few different languages but I choose Java. Just like the MapReduce programs, HBase requires a configuration, which contains the details of the HDFS cluster. The jar files for HBase are packaged in the lib directory similar to how hadoop packages its jars. The configuration will need to be pointed at the hbase-site.xml and it will need to know where the hbase master is located. To minimize time connected to the database, tables and column families can be created before establishing a connection to the database. A table must be made before a column family can be added to it. Column Qualifiers are made on the fly when inserting data. The most important part of making a table is the column families. They are difficult to change and require HBase to be restarted so it is advised to plan column families carefully. The Hbase site states that performance is impacted if to many column families are created, they suggest two or three column families. Since Hbase uses api calls, print outs can be used in between the calls, which is extremely useful when debugging. Connecting to Hbase is done by creating an HBase admin. The admin is just like an admin, administering the database through the cli but now through a Java program. I would suggest checking to make sure the master is running immediately after connecting, as this will let you know if there are any issues with the master. Upon connecting the admin can call create table and then close the connection. When making the table all of the column family information is stored in the table variable. This way only one call is needed. The creation can be verified by using the cli and issuing the “list” command.



Caption: Using the HBase shell, this shows that a table has been added.

Adding data:
Adding data to HBase is similar to creating a table it requires the same configuration but an admin is not needed. Since the table is already created and not being edited an admin connection is not necessary. The only command needed to place data in HBase is the “put” command. This command asks for a column family, column qualifier, and value. The table is specified when initializing the put class. One important thing about Hbase is, every value used in the “put” command needs to be passed as a byte array. HBase stores all its data as byte arrays, so when adding data it needs to be in the form of a byte array.

Caption: using the HBase shell, this shows all the contents of the SuperCars table.

MapReduce:
MapReduce using HBase is similar to normal map reduce but using HBase calls. So the main of the MapReduce looks slightly different. An HBase configuration needs to be created just like in the previous examples but here is where things change. In a normal MapReduce the mapper and the reducer need to be specified, the same happen but HBase has its own mapper and reducer calls. When specifying the mapper a scanner needs to be passed in for scanning the HBase data. This is also where you would specify what the mapper is emitting. If the reducer is writing to HBase the same approach is taken but if it is writing to HDFS then the reduce is treated like a regular map reduce program. The mapper accepts records as byte arrays so they will need to be parsed into whatever value they represent. Aside from the parsing, the mapper and the reducer act as they would in a regular map reduce program.

HBase thoughts:
HBase is a great framework for storing and searching through sparse data where not every column has value. There is a bit of learning curve due to it column oriented design.

Understanding the data model for HBase was by far the most challenging part for me. I originally worked with relational databases that are row oriented, so adapting to a new model, for how data is stored and searched, took some time. Once the user has a good comprehension of how data is stored it is relatively easy to use and has a lot of potential. If teaching a class on Hadoop I think that including HBase would be a good framework to teach and it would benefit students greatly.

Spark

Spark is an Apache framework for large scale data processing that differs from Hadoop in several ways. It can be run on a Hadoop cluster using Yarn to process data on an HDFS file system or it can be used on its own. Spark has the option to be run in standalone mode if a distributed file system is not provided. When running Spark on a Hadoop cluster, it maintains flexibility by being able to read data from many other frameworks, such as HBase, Hive, Cassandra and more. Spark sets itself apart from other data processing frameworks in the way it processes data. Spark loads everything into memory and utilizes Resilient Distributed Data sets (RDD) to maintain fault tolerance while processing the data. This is where the real power of Spark comes into play; because the data set is loaded into memory, multiple calculations can be performed on the data. When performing the same calculations with Hadoop, the data must be loaded each time from disk. Spark is written in Scala but has an interface for Python.

As mentioned above, Spark offers fault tolerance. When loading the data into memory Spark utilizes RDDs. Think of an RDD as a section of the data set that is loaded into a read only object. Each RDD knows where it received its data; this way if a RDD fails, it can automatically be rebuilt. Once data is loaded into memory, multiple calculations can be performed on the same data set. This functionality and speed is the basis for new frameworks being built on top of Spark. These frameworks are:

Spark SQL- essentially takes a NoSQL database and makes it faster.

Spark Streaming- provides stream processing on a Hadoop cluster.

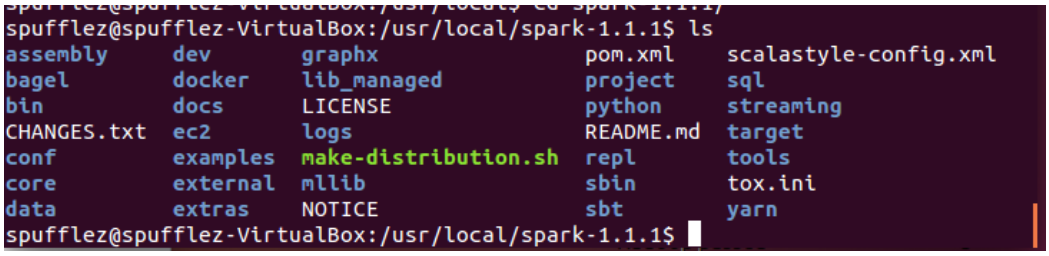
MLlib- is a machine learning library similar to mahout.

GraphX- is Spark's graphing library for graphing and performing graph calculations on a data sets.

Installing spark:

There are several installation options, pre built projects or build Spark from source. I choose to build Spark from source because I felt I could learn more about Spark. Also Spark can be run using HDFS as a data source, this is done using YARN or it can be run in standalone mode. I chose to run it in standalone mode, to see how it worked. Spark uses the local file system as it datasource, when running in standalone mode. To add nodes to the cluster, one

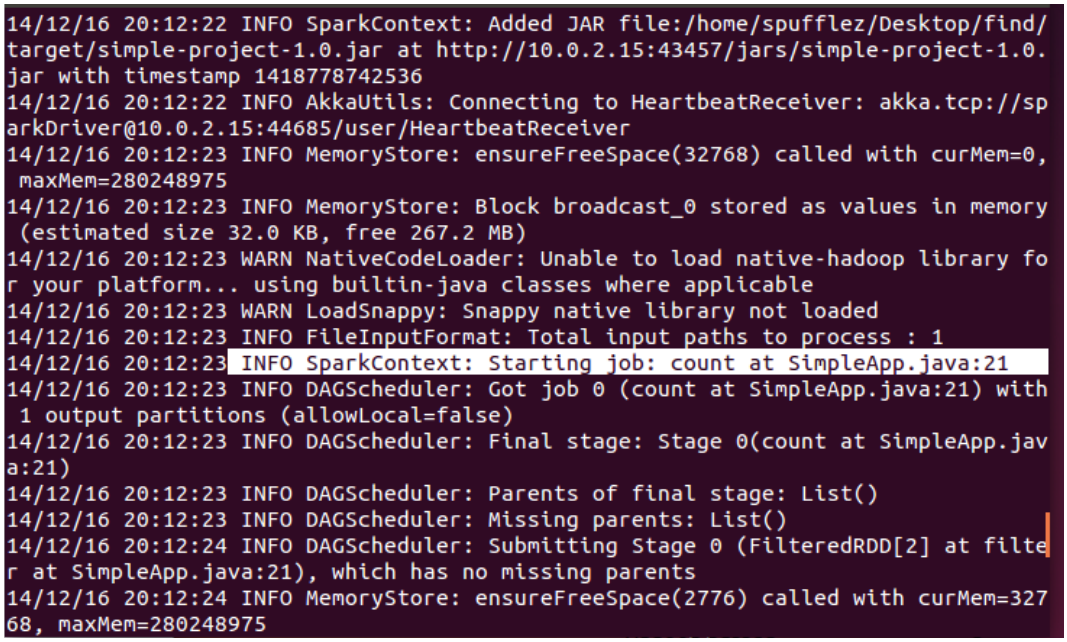
node is designated as the master and all other nodes just need Spark to be running and they will connect as slaves. To install Spark from source, upon downloading the tarball from there website, there is a readme file in the tarball. The only requirements are that java is installed and the JAVA_HOME variable is set. The readme contains the command to build Spark, which will take about an hour. If a dependence are not present while Spark is building it will throw an error and stop the build. This dependency will need to be installed to finish the Spark build. In my case I needed to install git and maven. The same command can be issued again to start installing Spark, and if the dependencies are installed Spark will resume right where it left off. When the build finishes Spark is ready for use.



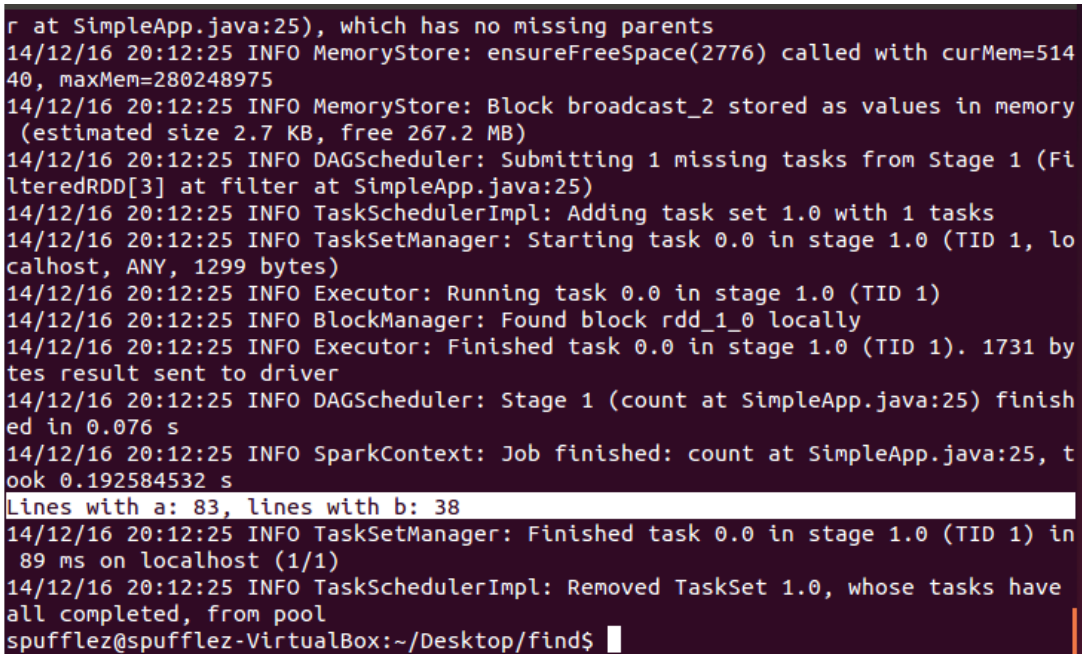
Caption: This shows the Spark directories.

Spark program:
Due to the semester ending I was only able to run the equivalent of a hello world program in spark but there were a few things that I learned. Unlike Hadoop and HBase which have a library containing all of these jar files Spark does not have a nice library. This may be different if a pre built installation of Spark is used but in my case only the java files were present. There is a built in Spark command that can be issued to to jar all the java files under each Spark directory.

Spark programs are similar to Hadoop programs in that they start with a Spark configuration. Since I am using Java, a java context needs to be created before creating an RDD. As mention above RDD's are used to store data, in this case an RDD is being used to store text data. After the RDD is created functions can be called to process the data. The neat thing about spark is the data does not need to be reloaded. Even in this simple example the same data is used twice without having reread it from disk. This is the major difference between Hadoop and Spark.



Caption: The highlighted part shows that Spark has started and it is initiating the job.



Caption: The highlighted line shows the output of the Spark job. The job was to count the occurrences of the letter “a” and “b” in a given text.

Spark thoughts:

The reason for studying spark is because it is a framework that challenges Hadoop. Due to the way it handles data it can process data in more efficient ways than Hadoop. In my opinion I think this is the trend industry is going to take, because it is faster and allows the user to process more data in less time. Spark also has a real time library as well, I did not get the

chance to explore it but because of this library it is one of the two frameworks that support real time processing. Storm supports real time processing as well but it is solely dedicated to real time processing unlike Spark which is more multitasked. Hadoop clusters will not go away anytime soon because of HDFS, but I think more companies will choose to run Spark on top of HDFS instead of Map reduce. The biggest down side to Spark is that it is memory dependent, the cluster can only perform as well as the max amount of memory available. That aside it is Hadoop's biggest contender and it was definitely worth studying.

Future Work:
Given more time I would like to further explore Spark as I feel it is the framework most companies will be using in the near future. Comparing the differences between Spark and Storm would be an excellent project. Sticking with Hadoop I would like to dive into the internal workings of Hadoop. Understanding how Hadoop works under the hood would allow for a greater understand of the framework, ultimately leading to an open source contribution to Hadoop. Making an alteration to how the framework runs would be the ultimate test proving my understanding of the frame work.

In conclusion I learned a great deal about Hadoop throughout this Independent study. Starting the semester I knew nothing about Hadoop, other then it was used to process big data. I now have a fully functional Hadoop cluster, and have run complex MapReduce programs on a 15 node Hadoop cluster. Much is being said about Hadoop but there is very little instruction being given as to how to use Hadoop. I think that RIT should teach Hadoop as part of a course or as a course in itself. Many companies are looking for students that know Hadoop and Hadoop like frameworks. That teaching Hadoop would greatly benefit the college and students. As I said before it is one thing to read about Hadoop and another to actually work on it.

Sources
Hadoop documentation
Apache. Hadoop - apache hadoop 2.5.0, Sept 2014.
http://hadoop.apache.org/. Accessed September 10, 2014.

HBase documentation
Apache. HBase- apache hbase 0.98.8, Nov 2014.
http://hbase.apache.org/. Accessed November 3, 2014.

Spark documentation
Apache. Spark- apache spark 1.1.1, Dec 2014.
https://spark.apache.org/. Accessed December 1, 2014.